SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 8, 357–387 (1996)

Research

Identifying Polymorphism Change and Impact in Object-orientated Software Maintenance

JERRY GAO, CHRIS CHEN AND Y. TOYOSHIMA
Fujitsu Network Transmission Systems, Inc., 2540 First Street, # 201, San Jose, CA 95131, U.S.A.

DAVID KUNG AND PEI HSIA
Department of Computer Science and Engineering, The University of Texas At Arlington, Arlington, TX 95019, U.S.A.

SUMMARY

Since polymorphism in object-orientated (OO) programming is an important feature and tool to increase the reusability and extensibility of object-orientated programs, understanding and identifying polymorphism changes and their impacts in an object-orientated program is very important in software maintenance. Although many published research articles have addressed the polymorphism feature in object-orientated programs, almost none of them have discussed software maintenance issues on identification of polymorphism change and impact. This paper proposes a formal software maintenance model for polymorphism in object-orientated programs, and discusses a systematic approach to track different polymorphism changes and their ripple effects. In addition, it reports on the implementation of a relevant tool, OOTME, and its use on two case studies.

KEY WORDS: software maintenance; object-orientated software technology; software change and impact analysis; object-orientated software maintenance; object-orientated programs

1. INTRODUCTION

Software maintenance is an important phase in a software life cycle. Software activities consume almost 70 per cent of the total life cycle budget. Reducing software maintenance cost is therefore extremely important (Freyd, 1993; Pressman, 1992, Lientz and Swanson, 1980, 1981; Schneidewind, 1987). As the object-orientated software technology enjoys its increasing acceptance in today's software industry due to its reusability, extensibility and understandability, OO software maintenance is becoming very important, and is considered as a challenging task. Although new features in the OO paradigm, such as classes, inheritance, overloading, polymorphism and dynamic binding, provide flexible tools for programmers to increase the reuse and understanding of programs, they bring new challenges or problems to software maintainers (Huitt and Wilde, 1992; Lejter, Meyers and Reiss, 1992; Crocker and Mayrhauser, 1993; Gao, 1995; Ponder and Bush, 1994).

OO program maintenance, like traditional program maintenance, consists of five basic

CCC 1040–550X/96/060357–31

Received 1 June 1996
Revised 5 September 1996

activities: (1) understand software specifications, and comprehend programs, (2) define and understand the needs of changes, (3) debug errors, such as locate errors and fix errors, (4) modify software specifications and change programs, (5) retest programs. The major problems in these maintenance activities are:

- Code comprehension is difficult for software maintainers particularly when they have a program written by other programmers. The new features in OO programs, such as overloading, polymorphism and dynamic binding are the major cause. First, these features complicate the understanding of real semantics of entities in an OO program (Lejter, Meyers and Reiss, 1992; Huitt and Wilde, 1992). For example, questions, such as 'which class function is invoked by this function call?' or 'which object member is referenced here?' are often asked by software maintainers. Next, polymorphism, like 'goto' in a program, can make a program difficult to understand if applied improperly (Ponder and Bush, 1994).
- Change identification and impact analysis in OO programs is a hard job. There are three factors: (1) complex dependence relationships between class objects and their members (Kung *et al.*, 1995a), such as inheritance, aggregation, and association, (2) overloading and polymorphism (Gao, 1995), which cause the difficulty in identifying the hidden changes and their impacts after program modifications, (3) dynamic binding, message passing, dynamic object creation and destruction, which increase the problems in tracing program changes and their impacts (Huitt and Wilde, 1992; Crocker and Mayrhauser, 1993).
- Current regression testing lacks adequate test criteria, good test models and effective test methods (Kung *et al.*, 1995a; Gao, 1995) because the existing regression test models test methods and test criteria do not consider these OO features, and do not address these change impacts.
- Debugging program errors is not an easy job due to these new features unless some systematic methods and powerful tools are available (Lejter, Meyers and Reiss, 1992; Huitt and Wilde, 1992).

It is clear that polymorphism and dynamic binding are two major causes of these problems. Although a number of ideas and methods have been proposed (Huitt and Wilde, 1992; Kung *et al.*, 1995a; Chen *et al.*, 1995) to address some of them, almost no systematic solutions are available to deal with change impacts of polymorphism and dynamic binding in OO programs.

Polymorphism in programming has been classified into different types (Cardelli and Wegner, 1985; Strachey, 1967). A number of formal systems (or type systems) have been developed to model and represent different types of polymorphism in programming languages (Abadi, Cardelli and Curien, 1993; Cardelli and Wegner, 1985; Freyd, 1993; Kortright, Levy and Montenyohl, 1993). These results are very useful for designing programming languages and their compilers, but not useful for software maintenance. Polymorphism and dynamic binding features in an OO program allow programmers to bind a program entity (such as an object pointer, or a call to an object function) to different program components based on the current status of a program and the given parameters. This does provide programming flexibility, but also causes a compiler or run-time support system to do some invisible binding job, which creates an understanding barrier between software maintainer and program.

This barrier causes the difficulties in understanding real semantics behind polymorphic program entities in a program. For example, an invocation of a virtual function in a class may correspond to different function implementations based on the program status and its class inheritance structure. Thus, identifying polymorphic features and their change impacts is a challenge job for software maintainers. To cope with these problems, a systematic method is needed to represent and identify *polymorphic entities*, such as polymorphic variables (or operators), and their corresponding components (such as a function definition, or a class definition), so that a support tool can be developed to help software maintainers in the following aspects:

- Identify and locate the presence of various polymorphic program components and entities.
- Enhance the code comprehension by identifying the real semantics behind these polymorphic entities.
- Trace changes of polymorphic components and their uses, so that the related impacts can be found.
- Provide an important basis for regression testing of polymorphism features.

This paper proposes a formal approach to representing different types of polymorphism in OO programs, and then presents a systematic method to solve the software maintenance problems caused by polymorphism. Using this method the presence of various polymorphic entities and their corresponding components can be automatically identified and located, and the real semantics of these polymorphic entities can be systematically revealed. Moreover, different polymorphism changes and their impacts can be traced. This information can be used to help regression testers define their test model and test criteria for polymorphism features, and conduct the regression tests. This method has been implemented at the Software Engineering Center For Telecommunications in the Department of Computer Science of the University of Texas at Arlington. The related implementation information, status, and the result data on case studies are reported in the paper.

The next section reviews the related work on OO software maintenance. Section 3 discusses polymorphism concepts and their challenges in OO program maintenance. In Section 4, a software maintenance model for polymorphism is presented. In Section 5, different polymorphism changes and their impacts are discussed, a systematic approach is described to identify them. Section 6 reports our implementation and case study. Finally, the concluding remarks are given in Section 7.

2. RELATED WORK

Although there have been a number of papers addressing software maintenance problems and support tools for OO programs, we lack concrete and systematic solutions to deal with the problems caused by polymorphism and dynamic binding features. In this section, we briefly review the existing work, and then relate our work to them.

Wilde and Huitt (1992) analysed problems of dynamic binding, complex object dependencies, dispersed program structure and control of polymorphism. They pointed out that these issues complicate high-level program understanding as well as detailed code comprehension. Some ideas and recommendations are made for possible tool support. Lejter, Meyers and Reiss (1992) also discussed the difficulty of maintaining an OO

software system due to the presence of inheritance and dynamic binding. They described their XREF/XREFDB prototype system that provided a text editing and relational database to support different queries from programmers to facilitate OO software maintenance.

Ponder and Bush (1994) assert that polymorphism can be considered harmful. They argue that like 'goto' statements, polymorphism can easily be abused to make an OO program hard to understand because: (1) in polymorphism, the same function call can refer to one of different function definitions (or implementations) of several classes; and (2) the choice is made at the compiler time (called static binding) or at the run-time (called dynamic binding) based on types of its parameters. Chen and Lee (1995) discussed several useful rules about how to achieve polymorphism safely by using genericity and inheritance in the right places. Following these rules, OO programmers can achieve the flexibility and expressive power of polymorphism in a safe manner.

Chen and Chang (1994) informally described their method for OO software maintenance. The basic approach is based on a maintenance database which consists of different schema representing object relationships at several abstraction levels. Based on this database, they highlighted their general ideas and simple procedures to perform different software maintenance activities without addressing polymorphism and dynamic features. Kung *et al.* (1995b) proposed a reverse-engineering approach to maintain OO programs at the code level. The authors discussed different types of code changes in an OO program, and provided a systematic method to conduct code change analysis and code impact analysis of a C++ program based on three abstract models extracted from the program. These results have been used in class regression testing to identify class firewalls to enclose all changed and affected classes (Kung *et al.*, 1995b; Gao, 1995). However, various polymorphism changes and their impacts are not considered in the method.

Opdyke (1992) in his thesis pointed out that changing OO software is harder than it might at first appear to be because changing OO software often requires changing the abstractions embodied in existing object classes and their relationships, or even changing the class structures. Tracking down class changes and their dependencies can be time consuming, difficult and error prone. In his paper, he defined a set of different refractorings (program restructurings) for OO programs, and proposed a systematic solution to deal with refractorings in an inheritance hierarchy (both generalization and specialization) and a class aggregation structure.

Crocker and Mayrhauser (1993) discussed the needs and the necessary software tools that support and enhance the software maintenance effort for OO programs. These include: (1) framework tools, such as a database and a displayer, (2) mundane tools, such as a parser, a graph and a symbol table, (3) knowledge tools, such as code browser and code slicer, and (4) change tools, such as ripple effect analyser and consistency checker. An early system for maintaining C++ programs was reported by Sametinger (1990). The system utilized the inheritance relation and containment relations (e.g., a class is contained in a file, or a method belongs to a class, etc.) to provide information text-based browsing facilities to an OO software maintainer. The C++ Information Abstractors (Chen and Grass, 1990) used program analysers to extract cross-reference information and store the information in a database. A maintainer could query the database to obtain the desired knowledge about a C++ program.

Unlike the existing work (Chen and Chang, 1994; Kung *et al.*, 1995b), where changes and impacts of various relationships between class objects are the major concern, this paper addresses the polymorphism feature in OO programs and its related issues on

change and impact analysis in software maintenance. A formal maintenance model is proposed to represent polymorphism features in OO programs. A systematic method is proposed based on this model to identify polymorphism changes and their impacts in OO programs. Unlike Opdyke William's work (1992), where OO program refractorings of inheritance and aggregation hierarchies are the major focus, the primary concern in this paper is how to identify different program changes related to polymorphism features and their impacts during software maintenance. In addition, the implementation of this method and two case-study results are reported here.

3. POLYMORPHISM CONCEPTS AND THEIR CHALLENGES

3.1. Polymorphism concepts

'Polymorphism' means the ability to take several forms to represent an entity (such as an object, or a function). In the object-orientated paradigm, there are a number of different definitions about polymorphism. For example, Booch (1990) in his book 'Object-oriented Design with Applications' defines 'polymorphism' as follows:

'A concept in type theory, according to which a name may denote objects of many different classes that are related by some super-classes; thus, any object denoted by this name is able to respond to some common set of operations in different ways.'

Rumbaugh *et al.* (1991) in their book 'Object-oriented Modeling and Design' give a simpler definition. They state:

'Polymorphism means that the same operation may behave differently on different classes.'

Meyer's definition about polymorphism addressed the dynamic aspect of polymorphism in OO programming (Meyer, 1989). He states:

"Polymorphism" means the ability to take several forms. In object-orientated programming, this refers to the ability of an entity to refer at run-time to instances of various classes. Polymorphism is the key to making the type system more flexible.'

Most traditional programming languages are known as monomorphic programming languages, such as PASCAL, in which parameters of each function (or operator) have a unique type. Each value and variable can be interpreted to be one, and only one, type. Unlike monomorphic programming languages, object-orientated programming languages (such as C++ and Smalltalk) are classified as *polymorphic programming languages* due to their polymorphism feature. 'Polymorphism' refers to the ability to take several forms. Unlike monomorphic programming languages, polymorphic programming languages consist of some variables and values which have more than one type (Cardelli and Wegner, 1985). In 1976, Strachey (1976) informally pointed out two kinds of polymorphism: *parametric polymorphism* and *ad hoc polymorphism. Parametric polymorphism* occurs when a function works uniformly on a range of types (which normally share some common structure). *Ad hoc polymorphism* occurs when a function works (or appears to work) on a number of different types in an *ad hoc* manner. There are two types of *ad*

hoc polymorphism, *overloading* and *coercion*. Overloading refers to the cases where a function name can be used to denote different function implementations, and the binding decision is made based on its context, including its qualifier expression (prefix expression), its actual parameters and their types. A coercion can be viewed as a hidden semantic operation which converts a function argument to its expected type of function. Coercions can be provided statically at the compiler time, or they can be supported dynamically by a run-time system. A detailed discussion about the differences between *coercion* and *overloading* can be found in Cardelli and Wegner (1985).

Cardelli and Wegner (1985) refined Strachey's view about polymorphism by introducing a new form of polymorphism, called *inclusion polymorphism* to model subtypes and inheritance in OO programming. In their view, both inclusion polymorphism and parametric polymorphism are classified as *universal polymorphism*. In *inclusion polymorphism*, an object can be viewed as an instance of many different classes (such as its super-classes). In *parametric polymorphism*, a polymorphic function has an implicit (or explicit) type parameter as an argument of that function.

The major differences between *ad hoc polymorphism* and *universal polymorphism* are:

- In universal polymorphism, a polymorphic function usually works on an infinite number of types which have some common structure, whereas in *ad hoc* polymorphism, a polymorphic function works on a finite set of unrelated types.
- In universal polymorphism, type arguments are used in parametric functions (called function templates in C++), however no type parameters are involved in *ad hoc* polymorphism.

From the implementation point of view, a universal polymorphic function shares the same code structure for arguments of any given admissible type, but an *ad hoc* polymorphic function may execute different code for each type of argument.

3.2. Polymorphic entities and their corresponding components

A *polymorphic program entity* in an OO program is an entity which can be referred to from more than one program component. There are two types of polymorphic program entities in C++ programs: *polymorphic function calls* and *polymorphic variables*. A polymorphic function call is a function invocation which can refer to more than one function implementation. A function invocation to a virtual function of a class in an inheritance hierarchy is a typical example. A pointer is known as a *polymorphic pointer* if it is used to point to more than one type of object (or data). A class pointer in a C++ program is a polymorphic pointer if it can point to different types of objects. Two types of inheritance are supported in current object-orientated programming languages. They are: *single inheritance* which means that a class in an inheritance hierarchy has at most one superclass, and *multiple inheritance* which means that a class in an inheritance hierarchy can have more than one superclass. For any OO program, an *inheritance hierarchy* can be defined to represent the inheritance relationships between a set of related classes.

Each polymorphic program entity has one or more program components which can be referred to it. A *polymorphic program component* is a program unit which may work, or appear to work, on different types of objects or arguments. There are three groups of

polymorphic components in C++. They are: (1) *overloaded functions* (or operators), (2) *polymorphic functions* (or operators), which include parametric functions (such as function template in C++), and generic function (such as virtual functions), and (3) *parametric class* such as class templates in C++.

An overloaded function in an OO program is a function (or operator) which shares its name with another function (or operator) in the same program scope. A C++ program may consists of overloaded global functions (or operators), overloaded friend functions (operators), and overloaded class member functions.

A parametric function in an OO program is a function consisting of a uniform code structure with different type arguments. A function template in a C++ program is a typical example.

A parametric class in an OO program is a class which consists of at least one parametric function member. A template class in C++ is a typical example.

A generic function of a class is a function which can be applied to different types of its derived class objects in a class inheritance hierarchy. For example, a virtual function of a polymorphic class in a C++ program is a generic function.

3.3. Polymorphism problems in software maintenance

Although polymorphism and dynamic binding provide programmers with a flexible and useful tool for software reuse, they cause some new problems in software maintenance. From a user point of view, different polymorphism types supported in OO programming languages may increase the understanding of OO programs at the user level because similar functions and type concepts are implemented using similar component names. However, this may cause the difficulty of code comprehension during software maintenance because these features encourage programmers to use identical names to refer to different implementations.

Edsger Dijkstra (Ponder and Bush (1994)) asserts that 'gotos' make a program difficult to understand because arbitrary changes of the program control flow by 'gotos' are difficult to follow in conjunction with dynamic changes to the state of the program data. Like 'gotos' in a program, polymorphism can be harmful to program understanding if it is not used properly Edsger Dijkstra (Ponder and Bush, (1994)). Similar to 'gotos', the binding relationships between polymorphic entities and their actual corresponding components can be arbitrarily changed because:

- Various binding actions performed by a compiler (or a run-time system) are hidden to program maintainers.
- The binding decisions are made based on variables (or object types), the program context, the corresponding inheritance hierarchy, and a set of binding resolution rules. The variable types may be determined dynamically based on the data state of the OO program.

These cause the code comprehension problems associated with polymorphic components and entities during software maintenance because software maintainers are not only concerned with the understanding of general concepts, but also with the following questions:

- Which function is actually invoked by a function call?
- Which object is referred to by any particular class pointer?
- What and where are the polymorphic program components and entities in an OO program?
- Which polymorphic components and entities have been changed after program modification? What are the ripple effects from the changes made?

According to our observations, we found that these questions are frequently asked by programmers and maintainers. Finding the answers to these questions is not only tedious but also difficult unless some systematic methods and supporting tools are available.

An OO program may consists of many polymorphic components or entities. Any changes (say deletions, updates and additions) on these polymorphic components or entities may cause invisible semantic changes and impacts. These changes and their ripple effects usually are easily found. Consider a class library which contains several groups of related classes. As shown in Figure 1, there are three class groups: X, Y and Z. Figure 1(a) shows that the class inheritance hierarchy of the class group Z consists of classes: A, B, C, D, E, F and H. Let us assume that two changes are made in class group Z: (see Figure 1 (b)) (1) add a virtual function $vf(x)$ and its implementation in class D, (2) delete the function $f(x)$ and its implementation from class C. Figure 1(c) shows the three invisible change impacts on $Foo{:}k1(x)$ in class group Y although there is no code change involved in this class group.

For software maintainers, these types of changes and impacts are hard to identify, trace and understand due to the following two reasons. First, these binding changes are acceptable by a compiler (or the run-time system), so no change information will be provided by the compiler and the run-time system. Therefore, they need to go through the source code to find these invisible changes. Obviously, this is a tedious job and prone to error. Secondly, after finding the changes, they need to figure out the updated binding relationship between a function call and its corresponding implementations. This is very difficult unless some support tools are provided. As pointed out in Lejter, Meyers and Reiss (1992) and Huitt and Wilde (1992) new systematic methods and support tools are needed to help maintainers deal with these problems.

4. SOFTWARE MAINTENANCE MODEL FOR POLYMORPHISM

To solve the polymorphism problems in a systematic manner, a formal model is needed to represent different polymorphism features, including polymorphic entities, corresponding components, and their mapping relationships (called polymorphism relationships) between polymorphic entities and components in OO programs. A systematic approach is proposed to identify different polymorphic components and entities, and their mapping relations, so that their changes and impacts can be easily tracked and identified.

4.1. Polymorphism

4.1.1. Predicates

In traditional programs, all referenced program variables (including function calls) are non-polymorphic because each of them has a unique interpretation and semantic. However,

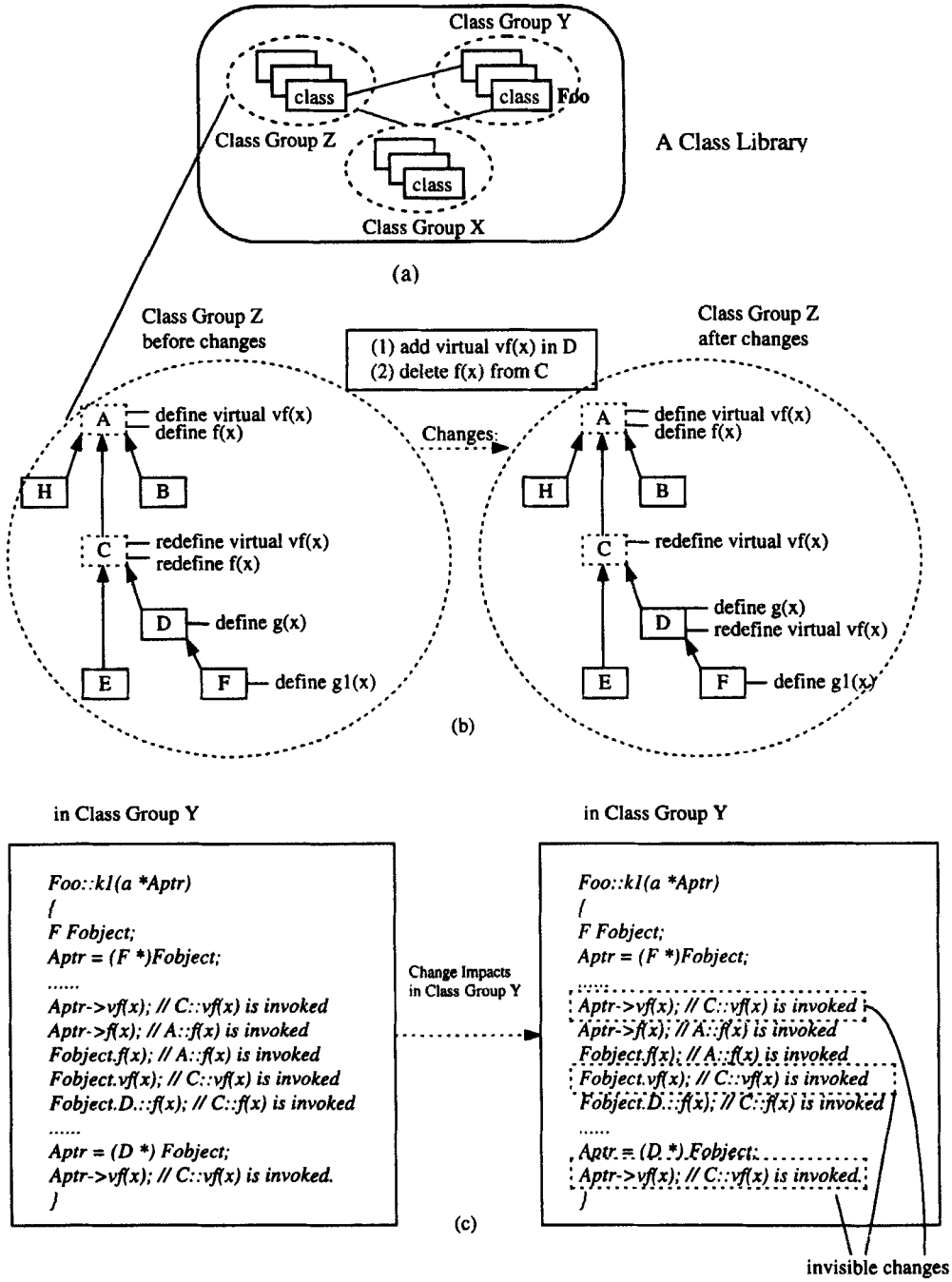


Figure 1. An example

current OO programs (such as in C++ and Smalltalk) may contain polymorphic variables or entities, which have more than one interpretation and semantic (see Figure 1). Their binding resolutions are based on a specific resolution rule, a program inheritance hierarchy, the function parameters and the program state at compile time or run-time.

Intuitively, polymorphism features in an OO program P can be considered as a relation between a polymorphic variable (or a polymorphic entity) in a program scope P_i and a set of program components which can be referred to. The binding resolution for a polymorphic variable (or an entity) can be viewed as a selection process which chooses only one program component as its referred component based on a set of resolution rules. In order to define a formal model representing different polymorphism features and their binding resolutions, a set of meta predicates are given below.

Let $C(P)$ be the set of all classes in an OO program P, $O(P)$ be the set of objects declared in P, $F(C)$ be the set of defined/redefined function members in class C, and $Fc(Pi)$ be the function call in Pi.

1. *ancestor*(C_i, Cj) = TRUE if class Ci is one of ancestor classes of class Cj, FALSE otherwise.
2. *classfmember*(f, Ci) = TRUE if the function f is defined/redefined in class Ci, FALSE otherwise.
3. *genericfunc*(F_i, Cj) = TRUE if (*classfmember*(Fi, Cj) = TRUE) and (function Fi is a generic function), FALSE otherwise.
4. *overloadfunc*(F_i, Cj) = TRUE if (*classfmember*(Fi, Cj) = TRUE) and (function Fi is an overloaded function defined in class Cj), FALSE otherwise.
5. *overloadop*(F_i, Cj) = TRUE if (*classfmember*(Fi, Cj) = TRUE) and (function Fi is an overloaded operator defined in the programming language), FALSE otherwise.
6. *EQname*(x, yj) = TRUE if the function call x has the same name as the function yj, FALSE otherwise.
6. *MatchFS*(x, yj) = *TRUE if the real parameters of the function call* x *match the signature of the function* yj, FALSE otherwise.

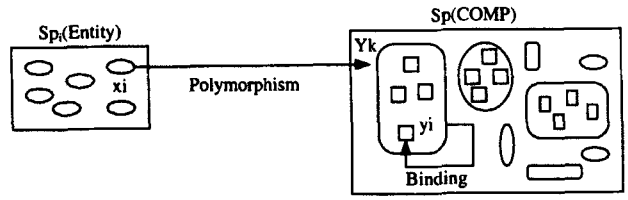
Figure 2 models different polymorphism features as the relationships between polymorphic entities and their corresponding components through a static (or dynamic) binding function. Their detailed definitions are given below. Figure 2(a) presents a general model for polymorphism feature.

4.1.2. *Definition: polymorphism*

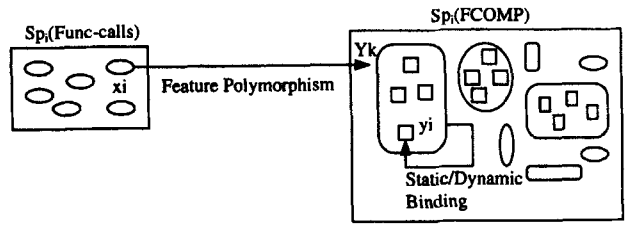
Polymorphism in a program scope P_i of P, is a 1–n ($n >= 2$) relation, denoted as $Rpoly(Pi)$, on $SPi(E) \times 2^{Sp(COMP)}$, where $S_{Pi(E)}$ is a set of referenced program entities in Pi, and $Sp(COMP)$ is a set of corresponding program components which can be referred to by a program entity in $SPi(E)$. The formal definition of $Rpoly(Pi)$ is given as follows:

where x is a referenced program entity in $S_{Pi}(E)$, Y ($|Y| >= 2$) is a subset of the program component set $Sp(COMP)$, and $P(x,Y)$ is a predicate which holds for x and Y. An element of $Rpoly(Pi)$ is denoted as $\langle x, Y \rangle$ which is an instance of the polymorphism relation.

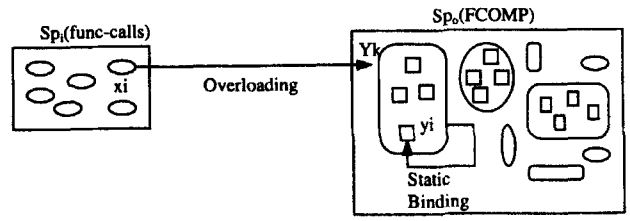
$R_{poly}(Pi)$ has the following two features:



(a) A General Polymorphism Model For OO Software Maintenance



(b) A General Model For Feature Polymorphism



(c) A General Model For Overloading

Figure 2. Polymorphism model for software maintenance

For any two elements $\langle x_i, Yi\rangle$ and $\langle xj, Yj\rangle$ in $Rpoly(Pi)$, if $xi = xj$, then $Yi = Yj$.
For any two elements $\langle x_i, Yi\rangle$ and $\langle xj, Yj\rangle$ in $Rpoly(Pi)$, if $Yi \langle\ \rangle\ Yj$, then $xi \langle\ \rangle\ xj$.

For any element $\langle x, Y\rangle$ of a polymorphism relation $R_{poly}(Pi)$, if $|Y| >= 2$, then x is known as a *polymorphic program entity*, and $\langle x, Y\rangle$ is called a *polymorphic relationship*.

4.1.3. Inclusion polymorphism

Inclusion polymorphism consists of two types: *inclusion object polymorphism*, and *inclusion feature polymorphism*. They are introduced into OO programs because of the inheritance relationship between object classes. Thus, both of them are related to inheritance hierarchies in OO programs. Inclusion feature polymorphism refers to the cases in which a function call of a class object in a program scope P_i can be possibly interpreted into several different function members (which share the same function name and formal signature) of its superclasses in a class inheritance hierarchy. The binding resolution is performed using a set of resolution rules according to its actual signature, the related

inheritance structure, and the program states at the invocation point. Figure 2(b) shows a general model for this relation.

Inclusion feature polymorphism in an OO program scope P_i is a $1{-}n$ ($n >= 2$) relation *Rpolyi(Pi)* on $SPi(Functcall) \times 2^{Sp(FCOMP)}$, where $S_{Pi}(Functcall)$ is a set of function prototypes appearing in P, and $S_p(FCOMP)$ is a set of functions defined in P. The formal definition of *Rpolyi(Pi)* in P is given as follows:

$$R_{polyi(Pi)} = \{\langle x,Y\rangle | (x \in S_{Pi}(Functcall)) \wedge (Y \in 2^{Sp(FCOMP)}) \wedge Pi(x,Y))\}$$

where x is a function call scheme appearing in P, Y is a subset of the generic functions in P, and $P_i(x, Y)$ is a predicate which holds for x and Y.

Let object O_k be the class object which receives a function call through x in the scope Pi, and class C_k be the type of O_k, $Pi(x, Y)$ is a conjunctive predicate consisting of the following items:

$(\forall yi)((yi \in Y) \wedge (EQname(x,yi)))$
$(\forall yi)((yi \in Y) \wedge (EQFS(x,yi)))$
$(\forall yi)((yi \in Y) \wedge (genericfunc(yi)))$
$(\forall yi)((yi \in Y) \wedge (\text{NOT}(Classfmember(C_k,yi))))$
$(\forall C_j)(\forall yi)((yi \in Y) \wedge (Classfmember(C_k,yi)) \wedge (C_i \in C(P)) \wedge Classfmember(C_j,yi)$
$\wedge ancester(C_j,C_k)))$

4.1.4. Overloading

There are three kinds of overloading in OO programs: *function overloading, operator overloading* and *type overloading*. Since an operator can be considered as a special case of a function, we only focus on function overloading and type overloading.

Function overloading refers to the cases where a function name is used by a function call in a program scope P_i to denote one of different overloaded function components. These overloaded function components share the same name and are defined in a common program scope, but they have different formal signatures and implementations. The resolution of a function overloading is performed based on its actual parameters. Figure 2(c) shows a general model for this relation.

Function Overloading in a program scope P_i is a $1{-}n$ ($n >= 2$) relation *Rpolyo(Pi)* on $SPi(Functcall) \times 2^{Sp(FCOMP)}$, where $S_{Pi}(Functcall)$ is a set of function prototypes which corresponding to the function calls appearing in *Pi*, and $Sp(COMP)$ is a set of functions defined in P. The formal definition of *Rpolyo(Pi)* in P is given as follows:

$$R_{polyo}(Pi) = \{\langle x,Y\rangle | (x \in S_{Pi}(Functcall)) \wedge (Y \in 2^{Sp(FCOMP)}) \wedge Po(x,Y))\}$$

where x is a function call scheme appearing in P_i, Y is a subset of overloaded functions (or operators) in $Sp(COMP)$, and $Po(x, Y)$ is a predicate which holds for x and Y.

Let object O_k be the class object which receives a function call through x in the scope Pi, and class C_k be the type of O_k. $Po(x, Y) = P_{o1}(x, Y)\ V\ P_{o2}(x, Y)$ is a predicate, where both $P_{o1}(x, Y)$ and $P_{o2}(x, Y)$ are predicates defined as follows.

$P_{o1}(x, Y)$ is a conjunctive predicate consisting of the following items:

$(\forall yi)((yi \in Y) \wedge (EQname(x,yi)))$

$(\forall yi)((yi \in Y) \wedge (Classfmember(C_k,yi)))$
$(\forall yi)((yi \in Y) \wedge (Overloadfunc(yi,Ci) \vee (Overloadop(yi,C_i))))$

$P_{o2}(x, Y)$ is a conjunctive predicate consisting of the following items:

$(\forall yi)((yi \in Y) \wedge (EQname(x,yi)))$
$(\forall yi)(\exists C_j)((yi \in Y) \wedge (C_j \in C(P)) \wedge Classfmember(C_k,yi) \wedge ancestor(Cj,Ck))$
$(\forall yi)((yi \in Y) \wedge (Overloadfunc(yi,Ci) \vee (Overloadop(yi,C_i))))$

This definition is applicable to overloading template functions if we consider overloaded template functions as a special type of function which consists of type arguments, because these template functions share the same name, but different type values to the common type arguments. Unlike a set of normal overloaded functions (which usually have different implementation structure), overloaded template functions usually share a common code structure.

Class overloading refers to the cases where a class template name is used in a program scope P_i to denote different overloaded template classes (which are instances of class template). These overloaded function components share the same function name, and they are defined in the same program scope, but they have different formal signatures and implementations. The resolution of class overloading is performed based on the actual parameters in an instance of class template. In C++ class overloading is resolved at compile time.

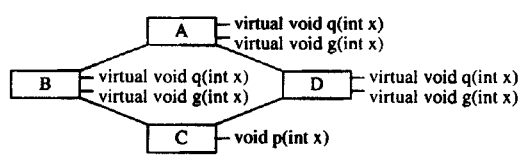
4.1.5. Polymorphism examples

Figure 3(a) shows a multiple class inheritance hierarchy consisting of four different classes: A, B, C and D. In class A, two virtual functions are defined. They are *void q(int x)*, and *void g(int x)*. Both of them have been redefined in class B. Class C has two virtual functions: *void f(int y)*, and *void k(int x)*. Unlike other classes, class D only defines a non-virtual function *void p(int y)*.

Consider the program scope P_i = *Otherclass::foo(B b, A *Aptr)* in Figure 3(b). There are four function calls: *x1' = b.g(intdata)*, *x2' = Aptr->q(intdata)*, *x3' = Aptr->p(intdata)*, and *x4' = Aptr->g(intdata)*. They correspond to the function prototypes: *x1 = b.g(int x)*, *x2 = Cptr->q(int x)*, *x3 = Aptr->p(int y)*, and *x4 = Aptr->g(int x)* respectively.

Because the function call x_1, is received by the object *b* and the corresponding set Y_1 consists of two generic functions, where Y_1 = {*A::void g(int y)*, *B::void g(int y)*}. The function call *Aptr->g(intdata)* is issued through the class pointer *Aptr* (its original type is class A). The actual object receiving this call is an object of class D. Because the function *g(int x)* is not defined in class D, but defined in its dominate class A and B, $\langle x_1', Y_1 \rangle$ is an element of the feature polymorphism relation *Rpolyf(Pi)* in the program scope *Pi* according to *Pi(x, Y)*. As shown later in Figure 6, there are three elements in *Rpolyf(Otherclass::foo(B b, A *Aptr))*. They are: $\langle x_1', Y_1 \rangle$, $\langle x_2', Y_2 \rangle$, $\langle x_3', Y_3 \rangle$.

Figure 4(a) shows a class inheritance hierarchy consisting of four different classes: A, B, C and D. In class A, two overloaded functions are defined. They are *void f(int x)*, and *void f(int x, int y)*. Similar to class A, both class B and class C have two overloaded functions. Consider the program scope P_i = *Otherclass::bar(B b, C *Cptr)* in Figure 4(b). There are three function calls: x_1' = *b.g(intdata)*, x_2'= *Cptr->k(intdata)*, and x_3'= *Cptr->p(intdata)*.




```
Otherclass::foo(B b,A *Aptr)
{
    int intdata;
    .....
    b.g(intdata);
    .....
    Aptr = new D;
    Aptr->q(intdata);
    Aptr->p(intdata);
    Aptr->g(intdata);
    .....
}                                   (b)
```

Function Calls:

x_1' = b.g(intdata)
x_2' = Aptr->q(intdata)
x_3' = Aptr->p(intdata)
x_4' = Aptr->g(intdata)

Generic Functions:

Y_1 = { A::virtual void g(int x), B::virtual void g(int x) }
Y_2 = { C::virtual void q(int x), C::virtual void q(int x) }
Y_3 = { D::void p(int y) },$|y_3| < 2$.
Y_4 = { A::virtual void g(int x), B::virtual void g(int x) }

Feature Polymorphism in the scope P, where P = *Otherclass::foo(B,b,A *Aptr)*

$R_{polyP}(P) = \{ \langle x_1', Y_1\rangle, \langle x_2', Y_2\rangle, \langle x_3', Y_3\rangle \}$

Feature Polymorphism Resolution:

$Bind_P(x_1', Y_1)$ = *B::virtual void g(int x)*
$Bind_P(x_2', Y_2)$ = *B::virtual void q(int x)*
$Bind_P(x_4', Y_4)$ = *B::virtual void g(int x)*

Figure 3. An example of feature polymorphism

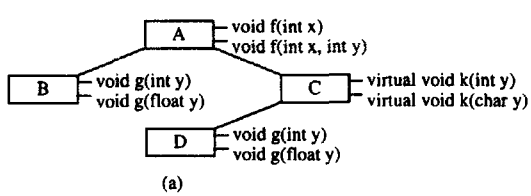
Because the function call x_1 is received by the object b and Y_1 = {*B::avoid g(int y)*, *B::void g(float y)*} is the corresponding set of overloaded functions, $\langle x_1', Y_1\rangle$ is an element of the overloading relation *Rpolyo(Pi)* in the program scope *Pi* according to P_{o1}. The function call *Cptr->k(intdata)* is issued through the class pointer *Cptr* (its type is class C). The actual object receiving this call is an object of class D. Since the function *k(int x)* is not defined in class D, but in its dominant class C, $\langle x_2', Y_2\rangle$ can be identified as an element of the overloading relation *Rpolyo(Pi)* based on P_{o2} where Y_2 is the set of overloaded functions. x_3' is a function call issued by the class pointer *Cptr*, and its associated object is an instance of class D. Since *void p(int y)* has been defined in class D as a non-overloaded function, there does not exist Y_3 corresponding to x_3'.

4.2. Polymorphism resolution

4.2.1. Rules introduction

To resolve a polymorphic entity x of $\langle x, Y\rangle$ in different polymorphism relations (for example, $R_{polyi}(P_i)$ or *Rpolyo*(P_i), a binding process is executed by the compiler (or the run-time system) to select one element (say y_j) from Y as its bound component based on a given set of resolution rules and the current program state. In this section, the detailed binding resolution rules are discussed for different types of polymorphism.

Let P_i be an program scope in P, *Rpoly*(P_i) be a polymorphism relation, and $\langle x, Y\rangle$ be one of its elements. Assume *Srule* = {*r*1, ..., *rn*} be a given set of resolution rules. The




```
Otherclass::foo(B b,C *Cptr)
{
    int intdata;
    .....
    b.g(intdata);
    .....
    Cptr->k(intdata);
    Cptr = new D;
    Cptr->p(intdata);
    .....
}
```

(b)

Function Calls:

x_1' = b.g(intdata)
x_2' = Cptr->k(intdata)
x_3' = Aptr->p(intdata)

Overloaded Functions:

Y_1 = { B::virtual void g(int x), B::virtual void g(float y) }
Y_2 = { C::virtual void k(int x), C::virtual void k(char y) }
Y_3 = { D::void p(int y) },$|y_3| < 2$.

Overloading Polymorphism in the scope P_i, where P_i = Otherclass::foo(B b,C *Cptr)

$R_{polyo}(P_i)$ = { $<x_1', Y_1>$, $<x_2', Y_2>$ }

Overloading Resolutions:

$Bind_p(x_1', Y_1)$ = B::void g(int x)
$Bind_p(x_2', Y_2)$ = C::virtual void k(int x)

Figure 4. An example of overloading

formal definition of a binding resolution, denoted as *Bind*(x, Y, S), for $\langle x, Y \rangle$ is given as follows:

$$Bind(x,Y) = (x,y_k)\text{if}(\exists r_i)(\exists y_k)((y_k \in Y) \wedge (r_i \in S_{rule}) \wedge r_i(x,y_k))$$

where S is the current status of P when a dynamic binding process is performed. In a static binding process, S can be ignored.

4.2.2. *Overloading resolution*

Binding resolution process for an overloading polymorphism is performed by the compiler, thus it is called static binding. Let P_i be a program scope in P, *Rpolyo*(P_i) be an overloading polymorphism relation, and $\langle x, Y \rangle$ be one of its elements. The formal definition of a static binding resolution, denoted as *Bindo*(x, Y), for $\langle x, Y \rangle$ is given as follows:

$$Bind_o(x,Y) = (x,y_k)\text{if}(\exists r_i)(\exists y_k)((y_k \in Y) \wedge (r_i \in So_{rule}) \wedge r_i(x,y_k))$$

where So_{rule} is a set of resolution rules for overloading polymorphism, the rule r_i is an element of *Sorule*, and $ri(x, y_k)$ is the application of x and y_k. Let $\langle x, Y \rangle$ in *Rpolyo*(P_i), O_i be the object which received the function call through x, and class C_i be the type of O_i. The binding resolution rules in *Sorule* are given below.

Let $R_{polyo}(Pi)$ be an overloading relation in *Pi*, $\langle x, Y \rangle$ be one of its relation instance,

and x be a polymorphic program entity. The binding resolution process will select one of the elements (overloaded functions or operators) in Y based on x's signature and a given set of resolution rules. Before explaining details of the rules, we first introduce two concepts: *a best matching function* and a *feature dominant class.*

For any $\langle x, Y\rangle$ in $R_{polyo}(Pi)$, yj in Y is a best-matching function if there is a best match between yj's signature and x's signature by checking every two corresponding arguments based on the matching steps in order (Stroustrup, 1994):

- Match them using no or only unavoidable conversions (for example, array name to pointer, function name to pointer to function, and T to const T).
- Match them using integral promotions (as defined in the proposed ANSI C standards; that is char to int, short to int, and their unsigned counter-parts) and float to double.
- Match them using standard conversions (for example, int to double, derived class object to base class object, unsigned int to int).
- Match them using user-defined conversions (both constructors and conversion operators).
- Match them using the ellipsis ... in a function declaration.

For any class object O_1 (its type is class C_1), if its function call scheme x appears in Pi and is referred to as inherited feature yk from a superclass C_2, then class C_2 is the feature dominant superclass of class object O_1 for yk if one of the following three conditions holds:

- In a single inheritance hierarchy, the class C_1 is the closest derived class of class C_2 which defines the feature y_k.
- In a multiple inheritance hierarchy, the class C_2 is the only superclass which defines y_k and matches the class qualifier in x. C++ actually supports a multiple inheritance hierarchy by a class qualifier in a function call to resolve ambiguity cases.
- In a multiple inheritance hierarchy where there is more than one superclass which defines a matching function member to x, the class C_2 is selected from these available superclasses according to a sequential order, for example, the left class is first.

Assume O_k be the object which receives x, C_k be the type of O_k.

Overloading rule 1. For any y_j in Y, if it is defined in C_k as an overloaded function (or operator), and it is a best-matching function (or operator) to x's formal signature, then $Bindo(x, Y) = (x, y_j)$ if y_j is a best-matching function (or operator) to x's formal signature in Y.

Overloading rule 2. For any y_j in Y, if it is not defined in C_k but it is an inherited function (or operator) of class C_k and y_j is defined as an overloaded function (or operator) for its feature dominant superclass from class C_k, then $Bindo(x, Y) = (x, y_j)$ if y_j is a best-matching function (or operator) to the x's formal signature.

4.2.3. Resolutions for inclusion polymorphism

The binding resolution process for the inclusion polymorphism is performed either at compile time or at run-time. Thus, the resolution is either a static binding or a dynamic

binding. Let P_i be an program scope in P, $Rpolyi(Pi)$ be an inclusion polymorphism relation, and $\langle x, Y\rangle$ be one of its elements. The formal definition of a static binding resolution, denoted as $Bindi(x, Y, S)$, for $\langle x, Y\rangle$ is given as follows:

$$Bindi(x,Y) = (x,Y_k)\text{if}(\exists r_i)(\exists y_k)((y_k \in Y) \wedge (ri \in Si_{rule}) \wedge r_i(x,y_k))$$

where S is the current status of program P, Si_{rule} is a set of resolution rules for inclusion polymorphism, r_i is an element in Si_{rule}, and $r_i(x, y_k)$ is the application of x and y_k.

4.2.4. *Resolution rules for feature polymorphism*

In feature polymorphism, there are two different types resolution rules: *static rules* and *dynamic rules*. Static rules are used by the compiler to resolve the polymorphic program entities. This type of binding process is called *static binding*. Dynamic rules are used at run-time to resolve the polymorphic program entities which can be resolved at compiler time. This type of binding process is called *dynamic binding*. For example, the function call *Object.f*(x) can be resolved at compiler time, and the function *Objectptr->*$f(x)$ will be resolved at run-time. In other words, which object is actually pointed to by the object pointer (*Objectptr*) can be determined only during run-time.

Let $R_{polyi}(Pi)$ be a feature polymorphism relation in Pi, $\langle x, Y\rangle$ be one of its polymorphism relation instances, and x be a polymorphic program entity. The binding resolution process will select one of the elements (generic functions or operators) in Y based on x's signature and a given set of the resolution rules. Assume O_k be the object which receives a function call through x, and C_k be the type of O_k.

Different function calls in P_i can be classified into two types: (1) a function call invoked through a base-class pointer, (2) a function call invoked through an object (or an object reference). In the following paragraphs, two sets of feature resolution rules are provided. They are: (1) static feature polymorphism rule, which is used for the function calls without the presence pointers, and (2) dynamic feature polymorphism rule, which is used for the function calls though a class pointer exists.

Feature polymorphism rule 0 (dynamic rule). For any y_j in Y, if it is not defined in C_k, but it is an inherited function (or operator) and defined in class C_i as a non-virtual function (or operator), then $Bindi(x, Y, S) = (x, yj)$ if the following four conditions hold:

- y_j is a best-matching function (or operator) to x.
- Class C_i is a feature dominant superclass of class C_k.
- O_k receives the function call x through a base-class pointer to C_i.
- The program status S holds for x in P.

Feature polymorphism rule 1 (dynamic rule). For any y_j in Y, if it is defined in C_k as a virtual function (or operator), and is a best-matching function to x, then $Bindi(x, Y, State) = (x, y_j)$ if x involves a class pointer to C_k and S holds for x in P.

Feature polymorphism rule 2 (static rule). For any y_j in Y, if it is not defined in C_k, but it is an inherited function (or operator) and defined in class C_i as a virtual function (or operator), then $Bindi(x, Y) = (x, y_j)$ if the following four conditions hold:

- y_j is a best-matching function (or operator) to x.

- C_i is a feature dominant superclass for class C_k.
- O_k receives the function call x by its name (or one of its references).
- The program status S holds for x in P.

5. POLYMORPHISM CHANGES IN OBJECT-ORIENTATED PROGRAMS

Change identification is one of the major tasks in regression testing and software maintenance. In current industry practice, change identification is only conducted by software maintainers in an *ad hoc* manner. There are many papers addressing this problem (Hartmann and Robson, 1989; Laski and Szermer, 1992). Their results can be used to identify different changes in traditional programs. In Kung *et al.* (1995a), various code structural changes in an OO program are discussed.

A class firewall concept is proposed to enclose all affected classes. In addition, a generation algorithm is provided to identify various class firewalls. However, polymorphism changes and their impacts are not considered in this approach. In this section, various types of polymorphism changes and their impact identifications are discussed. Figure 5

Scope	Basic Change Types
Global Scope	Add/delete/change an overloading function/operator Add/delete/change a template funciton/operator Add/delete/change a template class Add/delete/change a polymorphic class
Class Scope	Add/delete/change an overloading function/operation Add/delete/change a generic function Add/delete/change a template function Add/delete/change a friend function/operator
Function Scope	Change the body of an overloading function/operator Change the body of a generic function Change the body of a template function

(a) Basic changes of polymorphic components in C++

Code Changes
Add/delete a function incovation to a parametric function Add/delete an instantiation to a parametric class Add/delete a reference (or usage) of a polymorphic variable Add/delete a function invocation to an overloaded function Add/delete a function invocation to a virtual function

(b) Basic changes of polymorphic entities inside a function scope in C++

Figure 5. Different code changes on polymorphism features in C++

provides a classification of polymorphism changes in an OO program. These basic change types are explained as follows:

- *Polymorphic component changes.* Different polymorphic components (defined before) can be changed in the scopes of a function, a class (or a template class), a file, and globally. In a file scope (or a global scope), these changes include adding, deleting or changing a polymorphic component (such as an overloaded function/operator, a polymorphic class, a template class or a template function). In a class scope, these changes consist of adding, deleting or changing a polymorphic class member, such as a generic function, an overloaded function, a friend function or a template function. In a function scope, the changes refer to the changes of the body of a polymorphic function (or operator).
- *Polymorphic entity changes.* Changes of polymorphic entities (defined before) can appear only inside a function. These changes include: (1) adding or deleting a polymorphic function or invocation of a generic function (or an overloaded function/a function template), (2) adding or deleting a polymorphic variable, such as an object pointer or its use, and (3) adding or deleting a polymorphic type instance, such as a class template.
- *Polymorphism relation changes.* For any program scope P_i, changes of polymorphism relations (defined before) include: (1) adding a new polymorphism relation, (2) deleting an existing polymorphism relation, and (3) changing an existing polymorphism relation. For any polymorphism relation, its elements can be changed in three different ways. They are: (1) adding a relation instance into a polymorphism relation, (2) deleting a relation instance from a polymorphism relation, and (3) changing the set of polymorphic components for a polymorphic program entity. Figure 6 lists different types of code modifications which may cause these changes.
- *Polymorphism resolution changes.* Changes of polymorphism resolutions can also be classified into three cases: (1) adding a new polymorphism resolution because of a new polymorphic entity, (2) deleting an existing polymorphism resolution because of a deleted polymorphic entity, and (3) changing an existing resolution due to the addition (or deletion) of polymorphic components.

5.1. Identification of polymorphism changes

5.1.1. Five steps

For two different versions of an OO program, the identification steps of their code changes on polymorphism consist of the following steps (see Figure 7):

1. Polymorphism analysis.
2. Change identification of polymorphic components.
3. Change identification of polymorphic entities.
4. Change identification of polymorphism relations.
5. Change identification of binding resolutions.

Polymorphism Relation Changes	Code Changes
Add a new polymorphism relation Rpoly (Pi)	Add a program scope Add polymorphic program entities Add polymorphic program components
Delete a polymorphism relation Rpoly (Pi)	Delete a program scope Delete polymorphic program entities Delete polymorphic program components
Change a polymorphism relation Rpoly (Pi)	All above

(a) Different Change On A Polymorphism Relation

Polymorphism Relation Changes	Code Changes
Add an instance <x, Y> into Rpoly (Pi)	Add a polymorphic program entity Add a polymorphic program component
Delete an instance <x, Y> from Rpoly (Pi)	Delete a polymorphic program entity Delete a polymorphic program component
Change an instance from <x, Y> into <x, Y'>	Add a polymorphic program component Delete a polymorphic program component

(b) Different Change On A Polymorphism Relation Instance

Polymorphism Resolution Changes	Code Changes
Add a polymorphism resolution (x, y)	Add a polymorphic program entity Add a polymorphic program component
Delete a polymorphism resolution (x, y)	Delete a polymorphic program entity Delete a polymorphic program component
Change a resolution from (x, y) into (x, y')	Add a polymorphic program component Delete a polymorphic program component

(c) Different Change On Polymorphism Resolutions

Figure 6. Changes of polymorphism relations and resolutions

5.1.2. Step 1: polymorphism analysis

This step consists of five phases. In the first phase, different program components are classified, and various polymorphic components are identified. These include:

- Types and classes which are defined by the system or users. Among them parametric types, parametric classes or polymorphic classes are polymorphic components.
- Functions and operators which are defined by the system or users. Their definition scopes can be a class or a file. Among them overloaded functions (or operators), parametric functions (or operators), and generic functions are polymorphic components.

In the second phase, various program entities in a program scope (such a class member function) are identified and classified into the following groups:

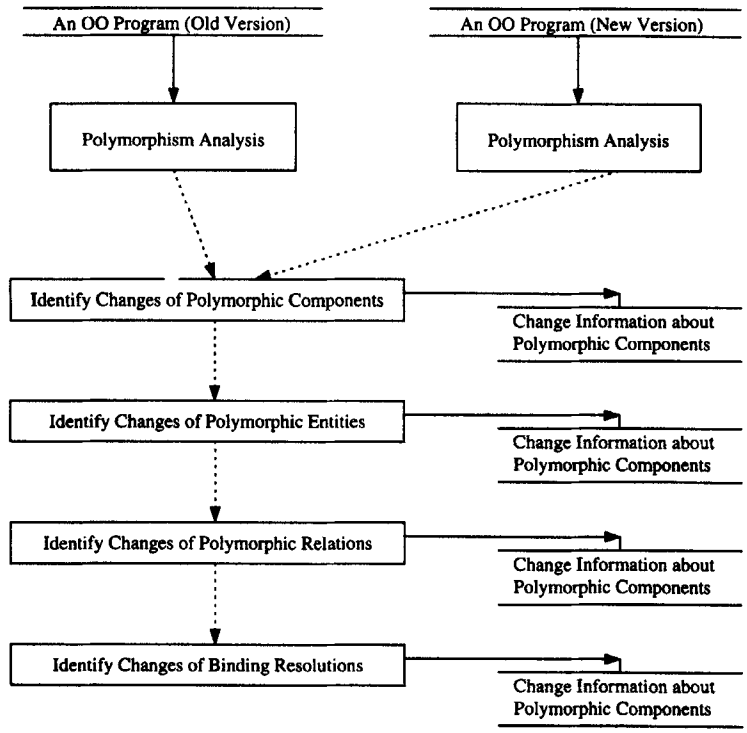


Figure 7. The change analysis process for polymorphism features

- declared program variables, which are the variables defined in a declaration statement, such as object variables, pointers and references, or simple data variables;
- key words and reserve words in the execution statements.
- operators or function calls, which appear in the execution statements;
- parametric function calls and parametric class instances appearing in the program scope;
- referenced program variables, which are the variables used in the execution statements, such as parameters, objects or data variables.

In the third phase, all polymorphic function calls (or operators) and polymorphic program variables in each program scope are recognized. Then, the related polymorphic relations are identified in the fourth phase. Finally, the binding resolution for each polymorphic program entity is recognized, based on the given resolution rules implemented in a specific OO programming language.

5.1.3. Step 2: change identification of polymorphic components

Any OO program P has its polymorphic component set $S_p(P) = T_p(P) \cup Fp(P)$, where $Tp(P)$ is a set of parametric classes, polymorphic classes, and parametric types, $Fp(P)$ is a collection of polymorphic functions in classes. These functions include virtual functions,

overloaded functions (or overloaded friend functions) and parametric functions. Notice that operators are considered as special cases of functions.

Let $S_p(P') = Tp(P') \cup Fp(P')$ be the polymorphic component set for a modified version (denoted as P') of the program P. Then, the different changes of polymorphic components can be identified as follows:

- If $(T_p(P) - Tp(P')) \langle\rangle EMPTY$, then any Ti in $(Tp(P) - Tp(P'))$ is a deleted polymorphic class, or a deleted parametric class (or type).
- If $(T_p(P') - Tp(P)) \langle\rangle EMPTY$, then any Ti in $(Tp(P') - Tp(P))$ is an added polymorphic class, or an added parametric class (or type).
- If any T_i in $Tp(P)$ and $Tp(P')$ is changed, then a residual polymorphic (or parametric) class is changed.
- If $(F_p(P) - Fp(P')) \langle\rangle EMPTY$, then any Fi in $(Fp(P) - Fp(P'))$ is a deleted polymorphic (or overloaded) function.
- If $(F_p(P') - Fp(P)) \langle\rangle EMPTY$, then any Fi in $(Fp(P') - Fp(P))$ is an added polymorphic (or overloaded) function.
- If any F_i in $Fp(P)$ and $Fp(P')$ is changed, then a residual polymorphic (or overloaded) function is changed.

5.1.4. Step 3: change identification of polymorphic entities

For any given program scope P_i, a set of polymorphic entities in Pi is denoted as $Ep(Pi) = PF \cup PV \cup PI$, where PF is a set of polymorphic (or overloaded) function invocations. PV is a set of references (or uses) of polymorphic program variables, such as object pointers, PI is a set of instantiations of parametric classes (or templates).

Let $E_p'(Pi) = PF' \cup PV' \cup PI'$ be the set of polymorphic entities in a modified version of Pi. Then different changes of polymorphic entities in Pi can be identified as follows:

- If $(PF - PF') \langle\rangle EMPTY$, then any Fc_i in $(PF - PF')$ is a deleted polymorphic (or overloaded) function invocation.
- If $(PF' - PF) \langle\rangle EMPTY$, then any Fc_i in $(PF' - PF)$ is an added polymorphic (or overloaded) function invocation.
- If any Fc_i in (PF and PF'), then Fci is a residual function call.
- If $(PV - PV') \langle\rangle EMPTY$, then any PF_i in $(PF - PF')$ is a deleted use (or reference) of a polymorphic variable.
- If $(PF' - PF) \langle\rangle EMPTY$, then any PF_i in $(PF' - PF)$ is an added use (or reference) of a polymorphic variable.
- If any PF_i in (PF' and PF), then PFi is a residual reference to a polymorphic variable.
- If $(PI - PI') \langle\rangle EMPTY$, then any PI_i in $(PI - PI')$ is a deleted instantiation of a parametric class.
- If $(PI' - PI) \langle\rangle EMPTY$, then any PI_i in $(PI' - PI)$ is an added instantiation of a polymorphic class.
- If any PI_i in (PI' and PI), then PIi is a residual instantiation of a polymorphic class.

5.1.5. Step 4: change identification of polymorphism relations

Assume $R_{poly}(Pi)$ is a polymorphism relation in a program scope Pi. Let $R'_{poly}(Pi)$ be the changed polymorphism relation. Then its changes can be identified as follows:

- If $R_{poly}(Pi)$ $\langle\ \rangle$ *EMPTY* and $R'_{poly}(Pi)$ $\langle\ \rangle$ *EMPTY*, then $R'_{poly}(Pi)$ is an added polymorphism relation.
- If $R'_{poly}(Pi)$ $\langle\ \rangle$ *EMPTY* and $R_{poly}(Pi)$ $\langle\ \rangle$ *EMPTY*, then $R_{poly}(Pi)$ is a deleted polymorphism relation.
- If $R_{poly}(Pi)$ $\langle\ \rangle$ $R'_{poly}(Pi)$, then any $\langle x, Y\rangle$ in $(R_{poly}(Pi) - R'_{poly}(Pi))$ is a deleted relation instance.
- If $R'_{poly}(Pi)$ $\langle\ \rangle$ $R_{poly}(Pi)$, then any $\langle x, Y\rangle$ in $R'_{poly}(Pi) - R_{poly}(Pi))$ is a deleted relation instance.
- If $R'_{poly}(Pi) = R_{poly}(Pi)$, then the polymorphism relation $R_{poly}(Pi)$ is not changed.

5.1.6. Step 5: change identification of binding resolutions

It is clear that the first four steps can be implemented through a program parser in a static approach. However, step 5 must be performed in a dynamic approach according to the test cases in a given regression test plan. In other words, a tester can select and execute the test cases from a given test plan to cover a code segment (including polymorphic entities) to find out their binding resolutions based on each test case.

Let P be a program, T_i be a test case in a regression test plan, and STi be the pre-state of P for Ti. Assume $Rpoly(Pi)$ is a polymorphism relation in a program scope Pi, and $R'_{poly}(Pi)$ be the changed polymorphism relation. Then, the changes of polymorphic entities and their binding resolutions in Pi can be identified as follows:

- If $R_{poly}(Pi)$ has one element with a polymorphic entity x (say $\langle x, Y\rangle$), and $R'_{poly}(Pi)$ does not, then x is a deleted polymorphism entity.
- If $R'_{poly}(Pi)$ has one element with a polymorphic entity x (say $\langle x, Y'\rangle$), and $R_{poly}(Pi)$ does not, then x is an added polymorphism entity.
- If $\langle x, Y\rangle$ in $R_{poly}(Pi)$ and $\langle x, Y'\rangle$ in $R'_{poly}(Pi)$, x's binding resolution is changed if, and only if, $Bind(x, Y, STi)$ $\langle\ \rangle$ $Bind(x, Y', STi)$.
- If $\langle x, Y\rangle$ in $R_{poly}(Pi)$ and $\langle x, Y'\rangle$ in $R'_{poly}(Pi)$, x's binding resolution is not changed if, and only if, $Bind(x, Y, STi) = Bind(x, Y', STi)$.

5.2. Polymorphism change impacts

A concept, called 'firewall', has been used in a number of research papers (Leung and White. 1990; Kung *et al.*, 1994) on software maintenance to enclose the changed and affected software components in a program, so that regression tests for its unchanged and unaffected components can be avoided at the unit level and the integration level. A control firewall notion is introduced in Leung and White (1990) to refer to an enclosure, which includes the affected modules and the control links among them after modifying a traditional program. A class firewall is used in Kung *et al.* (1994) to refer to an enclosure, which consists of the changed and affected classes after modifying a class in an object-orientated program. This section presents a systematic approach to identifying the impact of polymorphism changes by using a new firewall, called a *polymorphism firewall.*

As we know, after a polymorphic entity is added (or deleted), its related polymorphism resolution is also added (or deleted). A changed polymorphic entity can be considered the result of deleting a new entity or adding a new entity. Thus, the impact identification for a changed polymorphic entity is very simple. However, any change on a polymorphic component may cause various impacts on many different polymorphic entities. Here, we introduce a polymorphism firewall to enclose all affected (possibly) polymorphic entities after changing a polymorphic component. The detailed definition is given below.

A polymorphism firewall for a polymorphic component y in P is a set of polymorphic entities whose polymorphism relations, binding resolutions (or semantics) can be affected due to various changes of y.

There are two different approaches to generating a polymorphism firewall for a polymorphic component. The first approach is based on polymorphism relations in a program P. For a polymorphic component y, let $R_{poly}(Pi)$ be a polymorphism relation for one version of the program scope Pi, and $R'_{poly}(Pi)$ be the corresponding polymorphism relation of the modified version of Pi. The detailed computations of polymorphism firewalls are computed as follows.

If y is an added polymorphic component, then its polymorphism firewall in P_i, denoted as $APFW(y, Pi)$, is computed as

$$APFW(y,Pi) = \{x|(\forall \langle x, Y\rangle)((y \in Y) \wedge (\langle x, Y\rangle \in R'_{poly}(Pi)))\}$$

This firewall only indicates the impact of an added polymorphic component inside P_i. To compute its impact for the whole program P, the firewall can be computed as

$$APFW(y,P) = \bigcup_{\forall Pi \in P} APFW(y,Pi)$$

If y is a deleted polymorphic component, then its polymorphism firewall in Pi, denoted as $DPFW(y, Pi)$, is computed as

$$DPFW(y,Pi) = \{x|(\forall \langle x, Y\rangle)((y \in Y) \wedge (\langle x, Y\rangle \in R_{poly}(Pi)) \wedge (\langle x, Y - \{y\}\rangle \in R'_{poly}(Pi)))\}$$

This firewall only indicates the impact of a deleted polymorphic component inside P_i. To compute its impact for the whole program P, the firewall can be computed as

$$DPFW(y,P) = \bigcup_{\forall Pi \in P} DPFW(y,Pi)$$

If y is a changed (or residual) polymorphic component, then its polymorphism firewall in P_i, denoted as $CPFW(y, Pi)$, is computed as

$$CPFW(y,Pi) = \{x|(\forall \langle x, Y\rangle)((y \in Y) \wedge (\langle x, Y\rangle \in R'_{poly}(Pi)))\}$$

This firewall only indicates the impact of a changed polymorphic component inside P_i. To compute its impact for the whole program P, the firewall can be computed as

$$CPFW(y,P) = \bigcup_{\forall Pi \in P} CPFW(y,Pi)$$

The polymorphism firewalls generated above are the maximum firewalls for a changed (added or deleted) polymorphic component. The program entities inside each firewall are the possible entities which may be affected by the changes on this component.

Another approach to construct a polymorphism firewall is based on the polymorphism resolutions. Let $S_{PR}(Pi) = \{(x_1, y_1) \ldots\}$ be a set of polymorphism resolutions in Pi, where xi is a polymorphic entity and yi is a bounded polymorphic component. Then, the polymorphism resolution firewall, denoted as $RPFW(y, Pi)$, for a polymorphic component y in the program scope Pi is computed as follows:

$$RPFW(y,Pi) = \{x|(\forall \langle x, Y\rangle)((y \in Y) \wedge (\langle x, yi\rangle \in S_{RP}(Pi)))\}$$

This firewall only indicates the impact of a polymorphic component inside P_i. To compute its impact for the whole program P, the firewall is computed as

$$RPFW(y,Pi) = \bigcup_{\forall Pi \in P} RPFW(y,Pi)$$

6. IMPLEMENTATION AND CASE STUDIES

6.1. OOTME

In the previous sections, a systematic method is presented to help software maintainers (or regression testers) to identify polymorphism feature changes (including polymorphic component changes and polymorphic entity changes) and their ripple effects in OO programs. This method has been implemented as a subsystem in a system called OOTME (Kung *et al.*, 1995a), which stands for object-orientated testing and maintenance environment) at the Software Engineering Center for Telecommunication at The University of Texas at Arlington, Texas, U.S.A.

The subsystem, called polymorphism analysis, consists of five modules:

- a user interface module, which provides a friendly user interface between the system and software maintainers;
- a polymorphism identification module, which identifies various polymorphic components/entities, and their related polymorphism types in a given OO program or a class library;
- a change analysis module, which identifies various changes of polymorphic components/entities, and their polymorphism relations in a given OO program or a class library, including added/deleted/changed polymorphic components, entities and relations;
- a polymorphism resolution module, which consists of two parts: (1) a static binding process, which performs static bindings, (2) a dynamic binding process, which performs dynamic bindings based on a given test plan. The second part has not been implemented yet;
- a polymorphism firewall generator, which identifies the related polymorphism fire-

walls for each polymorphic component or entity in a program. The module will consists of two parts: (1) change identification of polymorphism components and entities from two different versions of an OO program, (2) impact identification of these related changes;

- a report generator, which produces a report about all polymorphic components and entities found in the given program.

The above modules (except the firewall generator) have now been implemented. Using the current subsystem, users can easily find out what and where are the polymorphic components/entities, their static bindings and related polymorphism relationships. We are planning to add a polymorphism firewall generator based on the firewall concept described before.

6.2. Case-study results

We have applied the subsystem to two different class libraries to conduct two case studies. The purpose of these case studies is to see what is the ratio of the polymorphic components and entities to non-polymorphic components and entities in these libraries. Table 1 summarizes the case studies in terms of counts in various categories.

In the first case-study, we applied the implemented subsystem on the two class libraries in the real world to see how many polymorphic components have been defined in these two class libraries, and what is their percentage.

The first class library is the InterViews class library. Figure 8 shows that the InterViews class library (version 3.0) has 122 classes, 91 of them include virtual functions, and 51 of them consist of overloaded functions or operators. These polymorphic components may cause code comprehension problems when we want to add, delete or update them. There are 61 classes, in which the ratio of the number of their virtual function members to their total number of function members is over 0·5. There are 19 classes, in which the ratio of the number of their overloading function members to the number of their total function members is over 0·5.

The second class library consists of 48 classes. Figure 9 indicates that the library has 44 template classes. In 40 of these classes, over 50 per cent of their function members are virtual functions. However, only five classes contain overloaded function members.

The results of these case studies show that the polymorphic components and entities are important parts of OO programs or class libraries. Understanding their changes and impacts is one important job for software maintainers. Using the proposed method, we can easily identify different polymorphic components and their uses in OO programs, their changes and impacts.

Table 1. Summary of the case studies

Function calls	Class	Static binding	Dynamic binding	Inclusion polymorphism	Overloading	Non-polymorphism
155	18	89	66	17	12	128

(a) Overloaded functions in classes

no. of overloaded functions/operators (in a class)	number of classes with overloading
0	70
1 - 9	43
10 - 19	6
20 - 23	3

(b) Virtual functions in classes

number of virtual functions/operators (in a class)	number of classes with virtual functions
0	31
1 - 9	69
10 - 19	12
20-29	4
30-41	6

(c) The percentage of overload functions in classes

x = percentage of no. of overloading members to its total function members	number of classes with overloading
x=0	70
0 < x <= 0.25	16
0.25 < x <= 0.5	17
0.5 < x <= 0.75	15
0.75 < x <= 1.0	4

(d) The percentage of virtual functions in classes

x = percentage of no. of virtual members to its total function members	number of classes with virtual members
x=0	31
0 < x <= 0.25	10
0.25 < x <= 0.5	20
0.5 < x <= 0.75	35
0.75 < x <= 1.0	26

(e) A statistics summary for all classes

total number of classes in the InterViews class library	122
total number of data members	397
total number of function members	1161
total number of virtual functions	655
total number of overloaded members	283
total number of overloading members	120

(f) A statistics summary for a class

average number of data members	3.25
average number of function members	9.52
average number of virtual functions	5.37
average number of overloading members	2.32

Figure 8. A statistical report for classes in the 'InterViews' class library

7. CONCLUSION

Software maintenance of OO programs is important. Polymorphism is one of the features which bring new issues to software maintenance, particularly in change analysis and impact evaluation. According to our case-study results, polymorphic components and entities are important reusable parts of OO programs. The identification of their changes and ripple effects is a difficult and error-prone task. To identify various changes and their impacts on polymorphic components and entities in an effective way, we need some systematic methods and automatic supporting tools. Until now, almost no published literature has addressed this subject.

In this paper, we addressed the maintenance problems caused by polymorphism features in object-orientated programs. A formal model is proposed to represent different polymorphism relations between polymorphic program entities and their components as well as their resolutions. Various changes on polymorphism features in OO programs are discussed, and a systematic approach to identifying these changes and their impacts is provided.

(a) Overloaded functions in classes

no. of overloaded functions/operators (in a class)	number of classes with overloading
0	43
1 - 5	5

(b) Virtual functions in classes

number of virtual functions/operators (in a class)	number of classes with virtual functions
0	4
1 - 9	26
10 - 19	8
20-29	2
30-45	8

(c) The percentage of overload functions in classes

x = percentage of no. of overloading members to its total function members	number of classes with overloading
x=0	43
0 < x <= 0.25	2
0.25 < x <= 0.5	3
0.5 < x <= 0.75	0
0.75 < x <= 1.0	0

(d) The percentage of virtual functions in classes

x = percentage of no. of virtual members to its total function members	number of classes with virtual members
x=0	4
0 < x <= 0.25	2
0.25 < x <= 0.5	2
0.5 < x <= 0.75	14
0.75 < x <= 1.0	26

(e) A statistics summary for all classes

total number of classes in a class library IX	48
total number of data members	8
total number of function members	764
total number of virtual functions	658
total number of overloaded members	19
total number of template classess	44

(f) A statistics summary for a class

average number of data members	0.17
average number of function members	15.92
average number of virtual functions	13.71
average number of overloading members	0.40

Figure 9. A statistical report for classes in the other class library

A subsystem, called polymorphism analysis, in the OOTME system has been implemented to identify polymorphic components, entities and various polymorphism relations between them in an OO program. Our application experience of this system indicates that it is very easy to find polymorphic components, polymorphic entities and their relationships using the system. In addition, it is very simple to find their changes and their impacts between two different versions of a given OO program. However, the current system does not consider the polymorphic entities involving dynamic binding resolutions. To solve this problem, a dynamic approach is needed to find real changes and their impacts based on a given test plan. We plan to complete the implementation of this part in the near future.

Acknowledgements

The material presented in this paper is based on the work supported by the Texas Advanced Technology Program (Grant No. 003656–097), Fujitsu Network Transmission Systems, Inc. and the Software Engineering Center for Telecommunications at the University of Texas at Arlington, Texas.

References

Abadi, M., Cardelli, L. and Curien, P.-L. (1993) 'Formal parametric polymorphism', *Theoretical Computer Science*, **121**(1–2), 9–58.

Booch, G. (1990) *Object-Oriented Design with Applications*, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA.

Cardelli, L. and Wegner, P. (1985) 'On understanding types, data abstraction, and polymorphism', *Computing Surveys*, **17**(4), 471–522.

Chen, J.-B. and Lee, S. C. (1995) 'Pursuing safe polymorphism in OOP', *Journal of Object-Oriented Programming*, **6**(2), 39–45.

Chen, J.-Y. and Chang, S. V. (1994) 'An object-orientated method for software maintenance', *Journal of Object-Oriented Programming*, **5**(1), 46–51.

Chen, Y.-F. and Grass, J. E. (1990) 'The C++ information abstractor', *Proceedings of C++ Conference*, San Francisco, CA, USENIX, Berkeley, CA, pp. 51–56.

Crocker, R. T. and Mayrhauser, A. V. (1993) 'Maintenance support needs for object-orientated software', *Proceedings of The Seventeenth Annual International Computer Software and Applications Conference (COMPSAC)*, Phoenix, AZ, IEEE Computer Society Press, Los Alimitos, CA, pp. 63–69.

Freyd, P. (1993) 'Structural polymorphism', *Theoretical Computer Science*, **115**(1), 107–129.

Gao, J. Z. (1995) 'A cost-effective regression testing methodology for object-orientated programs', Dissertation at the Computer Science Department of the University of Texas at Arlington, TX.

Hartmann, J. and Robson, D. J. (1989) 'Revalidation during the software maintenance phase', *Proceedings of the Conference on Software Maintenance—1989*, Miami, FL, IEEE Computer Society Press, Los Alamitos, CA, pp. 70–80.

Kung, D., Gao, J. Z., Hsia, P., Chen, C. and Toyoshima, Y. (1994) 'Identification of change impacts of object-orientated programs,' *Proceedings of International Conference on Software Maintenance—1994*, Victoria, Canada, IEEE Computer Society Press, Los Alamitos, CA, pp. 202–211.

Kung, D., Gao, J. Z., Hsia, P., Lin, J. and Toyoshima, Y. (1995a) 'Class firewall, test order and regression testing of object-orientated programs', *Journal of Object-Oriented Programming*, **6**(3), 51–65.

Kung, D., Gao, J. Z., Hsia, P., Toyoshima, Y., Chen, C., Kim, Y.-S., and Song, Y.-K. (1995b) 'Developing an object-orientated software testing and maintenance environment', *Communications of the ACM*, **38**(10), 75–87.

Laski, J. and Szermer, W. (1992) 'Identification of program modifications and their applications in software maintenance,' *Proceedings of the Conference on Software Maintenance—1992*, Orlando, FL, IEEE Computer Society Press, Los Alamitos, CA, pp. 282–290.

Lejter, M., Meyers, S. and Reiss, S. P. (1992) 'Support for maintaining object-orientated programs', *Transactions on Software Engineering*, **18**(12), 1045–1053.

Leung, H. K. N. and White, L. (1990) 'A study of integration testing and software regression at the integration level', *Proceedings of the Conference on Software Maintenance*, San Diego, CA, IEEE Computer Society Press, Los Alamitos, CA, pp. 290–301.

Levy Kortright, L. M. and Montenyohl, M. (1993) 'A type system for overloading and parametric polymorphism', *Proceedings of the ACM 31st Annual Southeast Conference*, ACM Press, New York, NY, pp. 88–95.

Lientz, B. P. and Swanson, E. B. (1981) 'Problems in application software maintenance', *Communications of the ACM*, **24**(11), 763–769.

Lientz, B. P. and Swanson, E. B. (1980) *Software Maintenance Management*, Addison-Wesley Publishing Co., Reading, MA.

Meyer, B. (1989) *Object-Oriented Software Construction*, Prentice-Hall, Englewood Cliffs, NJ.

Opdyke, W. F. (1992) 'Refactoring object-orientated frameworks', Dissertation in the Department of Computer Science at the University of Illinois at Urbana-Champaign, IL.

Ponder, C. and Bush, B. (1994) 'Polymorphism considered harmful', *Software Engineering Notes*, **19**(2), 35–37.

Pressman, R. S. (1992) *Software Engineering: A Practitioner's Approach*, McGraw Press, New York, NY.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. (1991) *Object-Oriented Modeling and Design,* Prentice-Hall, Englewood Cliffs, NJ.

Sametinger, J. (1990) 'A tool for the maintenance of C++ programs', *Proceedings of the Conference on Software Maintenance—1990,* San Diego, CA, IEEE Computer Society Press, Los Alamitos, CA, pp. 54–59.

Schneidewind, N. F. (1987) 'The state of software maintenance', *Transactions on Software Engineering,* **SE-13**(3), 303–310.

Strachey, C. (1967) 'Fundamental concepts in programming languages', *Lecture Notes for International Summer School in Computer Programming,* University of Copenhagen, Denmark, not paginated.

Stroustrup, B. (1994) *The Design and Evolution of C++,* Addison-Wesley Publishing Co., Reading, MA.

Wilde, N. and Huitt, R. (1992) 'Maintenance support for object-orientated programs', *Transactions on Software Engineering,* **18**(12), 1038–1044.

Wilde, N., Huitt, R. and Huitt, S. (1993) 'Dependency analysis tools: reusable components for software maintenance', *Proceedings of the Conference on Software Maintenance—1993,* Montreal, Quebec, IEEE Computer Society Press, Los Alamitos, CA, pp. 126–131.

Authors' biographies:

Jerry Gao is a senior member of technical staff at Fujitsu Network Communications, Inc. and works at the R&D software engineering group in Global Software Technology Division. He received his Ph.D. in Computer Science from the University of Texas at Arlington in 1995. His major research interests include software testing and maintenance methodology, object-orientated technology, testing and maintenance tools, software engineering support environment, world wide web technology and web-based application systems for software engineering.

Cris Chen is a director of the software engineering department at Global Software Technology Division in Fujitsu Network Communications, Inc. His current research interests include software engineering for object-orientated technology and software process management.

Yasufumi Toyoshima is a vice president of Fujitsu Network Communications, Inc. His current research interests include software engineering for object-orientated technology and software process management.

David C. Kung is an Associate Professor of Computer Science Engineering at the University of Texas at Arlington, Texas. He was an Assistant Professor of Computer Science and Engineering at the University of Iowa from 1987 to 1989 and a staff software scientist at International Software Systems, Inc. in 1990. Dr. Kung's research interests are in real-time systems and object-orientated systems. He received his M.S. and Ph.D. in Computer Science from the Norwegian Institute of Technology in 1980 and 1984, respectively.

Pei Hsia is a Professor of Computer Science and Engineering at The University of Texas at Arlington. He is also the director of the Software Engineering Center for Telecommunications. His research interests include requirements engineering, concurrent software engineering, incremental delivery, and software testing.